NASA
Technical Memorandum 106743

Army Research Laboratory
Technical Report ARL–TR–571

A Prototype Lisp-Based Soft Real-Time Object-Oriented Graphical User Interface for Control System Development

Jonathan Litt
Vehicle Propulsion Directorate
U.S. Army Research Laboratory
Lewis Research Center
Cleveland, Ohio

Edmond Wong
Lewis Research Center
Cleveland, Ohio

and

Donald L. Simon
Vehicle Propulsion Directorate
U.S. Army Research Laboratory
Lewis Research Center
Cleveland, Ohio

October 1994

National Aeronautics and
Space Administration

A Prototype Lisp-Based Soft Real-Time Object-Oriented Graphical User Interface for Control System Development

by

Jonathan Litt
US Army Research Laboratory
Vehicle Propulsion Directorate
Lewis Research Center
Cleveland, Ohio 44135

Edmond Wong
Advanced Control Technology Branch
NASA Lewis Research Center
Cleveland, Ohio 44135

Donald L. Simon
US Army Research Laboratory
Vehicle Propulsion Directorate
Lewis Research Center
Cleveland, Ohio 44135

ABSTRACT

A prototype Lisp-based soft real-time object-oriented Graphical User Interface for control system development is presented. The Graphical User Interface executes alongside a test system in laboratory conditions to permit observation of the closed loop operation through animation, graphics, and text. Since it must perform interactive graphics while updating the screen in real time, techniques are discussed which allow quick, efficient data processing and animation. Examples from an implementation are included to demonstrate some typical functionalities which allow the user to follow the control system's operation.

INTRODUCTION

When developing a graphical user interface (GUI) to be used in conjunction with a real-time control system, several factors must be considered and balanced in choosing a platform. Speed, graphics capability, networking ability, development environment, and cost are all important features to be weighed in the decision. Whether the user interface must interact with the control system or simply monitor it affects the choice of platform. In the first case, a hard real-time constraint is imposed which requires that the user always have access to the most recent data. In the latter case, when no immediate decisions are to be based upon the GUI's display, a soft real-time system will suffice by updating as time permits, allowing the user to see everything, but perhaps slightly delayed due to more urgent demands on the processor.

The graphical user interface developed for the Reusable Rocket Engine's Intelligent Control System (ICS) [1] at NASA Lewis Research Center is a soft real time, object-oriented, Lisp-based program. The platform chosen was a Texas Instruments Explorer II+ Lisp Machine which has a large, high-resolution, color screen, ethernet compatibility, a three-button mouse, and an extensive graphics support package containing functions ranging from primitives to detailed, highly complex development tools (figure 1).

The GUI is part of the ICS test bed (figure 2), a setup to demonstrate control systems in a laboratory environment [2]. The basic test bed consists of the AD100 simulation computer, the Control Interface and Monitoring (CIM) Unit [3], and the Explorer. The other computers in the figure are specific to the ICS and would not necessarily be used in other test bed setups. The test bed is designed so that the simulation computer could be replaced by the actual system being modeled and no major changes would be required in the connections or the remaining hardware and software.

In general, a closed loop system consists of a plant and a controller to regulate it. Depending on the complexity of the control system, other functions such as fault detection and sensor validation might be added but are not standard. In a research situation, however, the ability to monitor the closed loop system in some way is essential. The simplest method is to plot data using a strip chart recorder. A slightly less primitive method is to use a computer to capture the raw data and then plot it. While important capabilities as part of a larger monitoring system, by themselves these methods tend to limit the amount of data presented and thus the sophistication of the system itself. The relatively small number of channels available on the strip chart recorder and the inconvenience and delay associated with plotting data files make the testing and debugging of large systems difficult. Moreover, once the plots are created, the user is forced to interpret them off-line, out of the context of the real-time operation of the control system. For a relatively complex system such as the ICS, a detailed, customized, on-line interface is necessary to monitor the large number of variables and to allow the user to observe how the system performs under adverse conditions such as after component failures. An interactive GUI was developed because it enables the user to see, symbolically, what is going on inside the ICS through plots, animation, interactive graphics, and text.

AN OBJECT-ORIENTED GRAPHICAL USER INTERFACE

An *object-oriented system* is one consisting of entities possessing certain data and operations [4]. These entities or *objects* interact in predefined ways to give the overall system the desired qualities. By object-oriented graphics, one means a set of graphical entities which have certain properties. These properties might include position, color, and size, for instance. Items traditionally thought of as graphical objects—polygons, sprites, blinkers—are only a fraction of the total. Other graphical objects used in this GUI include windows, frames, and the mouse cursor. Specific instances of a class of objects are created from a template called a *flavor* which is a generic object of a particular type with certain default properties. The new object will automatically inherit those properties unless they are explicitly set to something else. New flavors can be built by combining several existing types and the new ones will contain the properties of their parents. Objects communicate by sending *messages* to each other which, when received, produce a certain action. For example, a sprite is a graphical object which can move along a desired path on the screen automatically, i.e., the tasks of saving the background, drawing the sprite, erasing the sprite, redrawing the background, saving the background in the new location, redrawing the sprite in its new location, and so on, are done by the processor without instruction from the GUI. These actions can be initiated and altered by sending a message to the sprite, which is accomplished by executing a special line of code. The sprite object itself contains the program to update the screen and has it simply by virtue of being a sprite—it is an

inherited property of all sprites.

THE BASIC FRAME

The GUI is made up of multiple screens which represent different aspects of the closed loop system a user may want to examine. Each screen consists of three windows built into a structure called a *frame*, an example of which is shown in figure 3. Since there exists a one-to-one relationship between screens and frames in this GUI, the words will be used interchangeably. The frames are arranged in a hierarchical, tree-like structure. The more general represent the base of the tree while the more specific represent its branches. Each frame contains at least one *icon*, a graphical object which symbolizes an action to be performed. In this case, clicking on the icon with the mouse exposes the frame corresponding to that icon.

One of the development tools on the Explorer is the Constraint Frame Editor [5]. This utility allows the user to divide the screen into windows with the mouse and then create and save the code used to construct this frame. The code can be edited for user-designed applications. The basic frame used in the ICS GUI was developed with the Constraint Frame Editor. It consists of three windows of different types. After the general contents of the frame were created in the Constraint Frame Editor, the resulting code was edited for each frame so that every screen of the GUI was comprised of an individual frame containing three unique windows.

The Mouse-Sensitive Graphics Window

The upper window on each screen is *mouse-sensitive* (see figure 3). This means that there are objects on the screen which become highlighted when the mouse cursor is on them and that clicking on them causes some action to be performed. When the mouse cursor is placed over the selectable object, a box appears around the object. Additionally, a text string indicating which frame will be exposed if the object is clicked on appears in the extreme lower left of the screen. Both the box and the text string disappear when the mouse cursor is moved off the object. In this case, clicking on the icon will bring up the screen which corresponds to it. The mouse-sensitive graphics window contains a picture of the system or component so it is clear which icons might be selectable. To make it even clearer, whenever a failure occurs, the picture of the malfunctioning component starts blinking. With this type of screen it is natural to select particular objects creating a smooth flow through the GUI to observe the closed loop system's operation interactively.

The Plotting Window

Another graphics window is located at the lower right of each screen (see figure 3). These plotting windows are not mouse-sensitive. They are animated to display the values of the system variables in strip-chart form updated in simulation time. Each of these windows can contain several sets of coordinate axes displaying preselected variables appropriate to the picture in the mouse-sensitive graphics window on the same screen. Each set of axes can display up to 361 data points. When the plot comes to the right edge of the time-axis, it shifts left half way (the point at the right edge moves to the middle of the time-axis) and continues plotting to the right. Time is displayed across the top of the window and is updated with each leftward shift. Variable name, units, and range are included by each set of coordinate axes (figure 4).

The plotting window assumes a user-specified number of samples per second will be transferred from the CIM Unit and sets itself up so that the time-axis corresponds to 361 of those time steps (the time-axis is 361 pixels long). Ideally, the time stamps associated with the data sets correspond one-to-one with the pixels of the time-axis. If the GUI does not receive variable values for each expected time step, either because samples are sent at a slower rate or because some data packets get lost, the GUI will plot the points at their appropriate location based on the time stamp and linearly interpolate between the previous and current data point in the plotting window so spaces are not left blank. If the GUI receives samples at a faster rate than anticipated, implying that more than 361 data sets are received in the time allotted for 361, the array which saves the variable values will fill up prematurely causing a fatal error.

The Output Window

The output window is a Lisp Listener, an interactive text window located in the lower left corner of the screen (see figure 3). System bulletins such as announcements of failures are broadcast to all output windows for informational purposes. The output window accepts keyboard input, evaluates it and returns a value. Thus it can be used to examine or change variables within the GUI. To be able to type into the window, it must be selected with the mouse, i.e. the user must click on it.

Adding Screens

The GUI is modular. There are pieces of code corresponding to the creation of each frame. Copying the code and making minor changes such as to the names of the windows is all that is required to construct a new screen. To be able to expose the screen, it is necessary to put an icon in the mouse-sensitive graphics window of an existing screen which, when clicked on, will bring up the new frame. Once the window is created, a picture containing mouse-sensitive icons can be drawn in the upper window, and axes can be inserted in the lower right window.

Using the existing code as a guide, it is straightforward to create additional screens. Thus the GUI can be extended as needed without a great deal of effort. The appendix contains the code used to create the frame (but not any of the graphics) shown in figure 3. Developing the detail required in the picture in the mouse-sensitive graphics window might entail a significant amount of work.

PROGRAM FLOW

Because of the large, complex structure and detailed graphics comprising the GUI, the startup procedure is time-consuming. During this procedure, the screens and the graphical objects which they contain are created and built, as are the other objects and data structures used in the operation of the GUI. Once the startup operation is complete, the GUI waits for a network connection to be established by the CIM Unit. After the connection is made, the GUI loops continuously, reading data from the CIM Unit and using this information to update the screen (figure 5). User inputs from the mouse or keyboard will temporarily interrupt this looping. Unless the user input causes a catastrophic error, program flow will continue immediately following the GUI's response. Mouse inputs are explicitly checked for in the loop but keyboard inputs are accepted whenever they are typed. Generally, the time required to make simple

responses, such as changing or returning the value of a variable, will not be noticeable to the user.

Networking

The Explorer can communicate over ethernet, using the TCP/IP protocol. It is well known that ethernet is not generally acceptable for real-time communication because of the open-ended delay time associated with it. However, in the case of the Reusable Rocket Engine ICS GUI this problem was minimized by having only two processors on the network, the Explorer and the CIM Unit. This, along with the fact that the communication was one-way (from the CIM Unit to the Explorer) allowed it to be reliable and collision-free.

At regular intervals, the CIM Unit transmits a data packet to the Explorer which includes a time stamp, variable values to be displayed, and an integer indicating the failure status of each piece of hardware. This *data snapshot* provides all of the essential information about the ICS at a particular instant of time.

A critical aspect of network programming is resource allocation. When the Explorer reads the input buffer, the receiving routine creates an array to store the incoming data packet. Each time a packet is received, it is copied from the newly created array to an array which the GUI can manipulate. The created array is not reused and not automatically deallocated. Thus, after a time, these arrays could use up all of the free memory and crash the computer. Therefore, it is required that the DEALLOCATE-RESOURCE function be called after each packet is read.

Failures

When the CIM Unit detects a failure, a flag indicating its occurrence is sent over the network along with the data snapshot. Once received, the flag sets off blinkers in the mouse-sensitive graphics windows and causes messages to be printed in the output windows of the GUI.

The blinkers used are *bitblt-blinkers* which are raster images whose status may be :off (not visible), :on (visible), or :blink (alternately visible and not visible). Each blinker is created by rasterizing the image of a graphical object intended to blink. This blinker, whose status is initially :off, is inserted directly on top of the graphical object it resembles. When the appropriate failure flag is set, the blinker's status is set to :blink so the object appears to alternate rapidly between normal and abnormal coloration, caused by the *alu function* (the arbiter of the resultant color when foreground and background colors combine), giving the impression of flashing.

Since the closed loop system and the GUI are designed to work together in the test bed and the detectable failures in the system are known, it is possible to set up a failure signaling system between them. In the case of the Reusable Rocket Engine, there are 19 identifiable failures. The existence of each failure is represented by a logical flag, which, in turn, is represented by a binary digit, 0 or 1. The failures are ordered so that the sequence of failure flags can be represented as a 19-bit integer. Since the CIM Unit sends a packet of real numbers over the network, it must convert the failure flag integer to a floating point value which is included as the

last element of each packet. Once received, the floating point number is converted back to an integer which is then compared to the previously received integer using a logical XOR operation. A nonzero result triggers a series of tests for individual nonzero bits. Each one causes blinkers to begin flashing and failure messages to be broadcast to all output windows. In general, signaling from the CIM Unit to the GUI can be accomplished for any number of prearranged events. If the number of possible events exceeds the number of bits in a single integer, additional integers can be added to the end of the data packet which is passed to the GUI. The number of flags represented by each integer is limited by the internal representation of integers and real numbers and the accuracy of the conversion between them.

Operational Modes

The GUI is capable of operating in networked or stand-alone mode. The system was designed and is intended to run networked as part of the ICS testbed. However, during the development, testing and debugging phases, it is critical that the GUI have the ability to run on its own, duplicating its networked behavior. The user chooses the mode at start-up by calling the GUI routine with a logical argument indicating whether or not the system is networked. If the GUI is networked, the data values are read directly from the input buffer. If not, they are set within the program to some predetermined values. The two different data acquisition functions are the only code which is not common to both modes. The stand-alone mode allows a user to check every aspect of the GUI except for the reading of the input buffer. This way, all changes can be tested and evaluated without requiring the use of the rest of the testbed system.

An automatic reset option was built in to avoid having to stop and restart the GUI and CIM Unit each time the control is reset to nominal conditions before a new simulation run. To prepare for the reinitialization of the simulation, a button on the CIM Unit is pressed, causing the time stamp value sent over the network to the TI Explorer to be reset to 0.0 and remain at that value as long as the button is depressed. When a time stamp value of 0.0 is first received by the GUI, the screens are all reset and all flags and internal variables reinitialized. The GUI loops, reading the buffer, mouse and keyboard input as usual but does not plot in the plotting window nor save past data values until the first nonzero time stamp is received. Thus, only the data which are considered valid are plotted. This feature is also implemented in the stand-alone mode since the two modes are identical except for the data acquisition portion. When running stand-alone, the user can set a logical flag from any output window which will set and hold the time-stamp value at 0.0 and, likewise, the user can reset the logical flag to restart the progression of time. A block diagram of this automatic reset procedure is shown in figure 5. Note that as long as the time-stamp is 0.0, the previous data values are overwritten with the new set of variable values as the program loops. This way, once time starts advancing, the values plotted corresponding to a time-stamp of 0.0 are the ones received most recently.

REAL-TIME CONSIDERATIONS

Eliminating Overhead

Any real-time system should be able to execute code efficiently. Lisp Machines are specifically designed to run Lisp code quickly. The Explorer is not as adept at running graphics routines as some specialized graphics computers but does an acceptable job for the current application.

Thus, it is necessary to program in such a way that the negative aspects of Lisp are minimized.

memory management

Lisp is notoriously cumbersome, requiring a tremendous amount of overhead and creating an immense quantity of *garbage* (the dynamically allocated memory not available to any executable code because the pointer to it is lost [6]). Eliminating garbage totally is nearly impossible, but finding ways to reduce it is good programming practice and essential for real time applications. Like most reasonably powerful programming languages, Lisp allows the user to write subroutines with local variables and to pass arguments. Additionally, Lisp contains many constructs in which variables are *bound* or temporarily set to a value and released at the end of the function. Sometimes it is convenient to use these types of expressions but the allocation and deallocation of memory involved with temporary binding is time-consuming and it is wise to avoid it if possible. A good way to get around this problem is to use *global variables* wherever possible. A global variable is defined in advance outside a specific function so it exists in memory and any function can access it.

drawing methods

Associated with each window is a *world*. A window provides a view into the world. Each world has information about itself including a *property-list* which contains the name of every object in that world. When a new object is placed into the world using the :insert method, its name is added to the property-list and, if it's location is suitable, it will appear in the window when either the :draw or :refresh method is used. The object remains in the world until it is explicitly deleted from the property-list. In contrast, there are :draw methods which merely write a picture on the window without inserting a graphical object into a world. Thus, when the window is refreshed, the image is lost. This is not only faster but also extremely convenient for situations where the screen should be wiped clean regularly. Using the most basic :draw methods can result in even faster execution of the graphics routines. Generally, screen output is one of the slowest procedures anyway, and, coupled with the fact that the drawing methods can be extremely long and complex because of multiple consistency checks, drawing becomes a time-consuming operation. Since the source code for all methods and functions is readily available, it is simple to inspect it to determine what is the most basic drawing function and call it directly, thus bypassing the expensive overhead. In the plotting windows of the GUI, the primitive :draw-clipped-line method was used instead of the much more complicated :draw-line or :draw-polyline. Since the data were preprocessed anyway, there was not much likelihood of any of the data causing trouble for the plotting routine. Because both the time values and the plots displayed are temporary and must change every time the time-stamp exceeds the right end of the time-axis, sending the plotting window the :refresh message erases everything that is not in that world's property-list. Only the axes, variable names, ranges and units remain. The new time values as well as the left half of each plot are immediately drawn in the window and the right half of the plots, the new values, are drawn as they are received from the CIM Unit.

Data Transfer

As the data transfer rate is increased, a trade-off develops between the need to plot sufficient data points to show the response in detail, and the need to keep the display on the screen long enough

to see it properly. Another consideration is that if the data transfer rate becomes too high, the Explorer might not be able to read all of the data packets and lose some when the input buffer overflows. This is especially likely to happen when a new screen is selected using the mouse since there is a lot of software overhead associated with exposing a different frame. Exposing a new screen might take a second or two while the CIM Unit continues to send data. Because of the linear interpolation feature of the plotting window, even if some data are lost, the remaining points are connected by straight lines which masks the fact that some are missing. During transients, however, this has the potential to be misleading because the sampling process filters out some high frequency information. For the ICS example, the reusable rocket engine simulation is slowed down to run ten times slower than real time in order to accommodate some of the slower ICS hardware and software. The CIM Unit sends data over the ethernet to the GUI at a rate of 20 packets per second (real time) which is sufficient to display the transient plots of the slowed down ICS simulation. With this transfer rate, the graphs which appear in the plotting window are displayed for nine seconds (180 points divided by 20 points per second equals 9 seconds) between each leftward shift.

Note that the user-specified data transfer rate corresponds to the simulation time rather than real time if they are different. It allows the time axis to be set up to plot the appropriate number of data points based on the time stamp value. That is why, in figure 4, half of the time axis is 0.9 seconds but is displayed for 9.0 seconds.

It is important to select the data transfer rate to the GUI which allows the important frequencies to be displayed, and balance this against the ability of the Explorer to receive data packets without its buffer overflowing. If the data transfer frequency is low enough that the plot can be read easily, the transfer rate should be well within the range the Explorer can accept without a problem.

Array Representation

In Lisp there are many data *types*. A type refers to the data structure, amount of memory required and properties of the datum. Examples include lists, atoms, and arrays. A powerful feature of Lisp from a programming point of view but a serious drawback for real time execution is that variables need not be declared a particular type; thus a variable can be set equal to an array and later set equal to an atom. If the data type is not known in advance, memory cannot be preallocated as with traditional languages, it must be allocated dynamically as data structures are created. Additionally, since the variable's type can change, the variable must point to its value rather than contain it (indirect addressing). In fact, each variable has several *cells* associated with it to help interpret its value. The cells hold information about the variable such as location in memory, type, etc. A *locative* in Lisp is an object which points to a cell and is used to access a variable. If a variable has associated locatives, it means that there is a lot of overhead involved with retrieving its value. Worse, when an array is declared, locative cells are created to describe each of its elements.

numeric

Efficiency can be greatly increased through knowledge of an array's contents. If the array will

only hold numbers of a particular type (integers, 32-bit reals, etc.), and it is declared that way, the exact size is known. Thus, the memory will be preallocated and can be accessed directly without locatives. This saves both memory and overhead. In the ICS GUI, the large array used to store the received data snapshots for plotting purposes was declared this way since the time stamps and dependent variables are all real numbers.

offset

A large rectangular (two-dimensional) array (number of variables × 361) is used to store the data snapshots as they are received (figure 6). The values are copied, as a column, into the first free column of the array. There are situations where it is convenient to manipulate a large array containing all measured variable values. On the other hand, sometimes it desirable to work with a sequence of only a single variable's values. From the programmer's point of view and from a readability standpoint, the use of *offset arrays* simplifies the job of accessing data by defining one-dimensional arrays corresponding to individual rows of the large data array. An offset array is an array defined to overlap some of the same contiguous memory locations as a previously defined array. These row arrays allow the programmer to deal with a sequence of data points for a single variable using its name rather than having to manipulate a large array of many variables using indices.

Offset arrays may also be used to convert from one data type to another implicitly. In the GUI, the data packet from the CIM Unit is read in and saved, element-by-element, as an array of 8-bit bytes. In reality, however, the data transferred are single-precision real numbers, each four bytes long. The problem of accessing the variables is not in reading their values but, rather, knowing how to interpret them. Thus, an offset array of single precision real numbers was defined at the same location in memory as the four-times-as-many-elements-long byte-array into which the data are copied (figure 7). This definition carries with it all of the information on how to interpret the array elements. Therefore, reading the first element of the real-array returns the four-byte-long, single-precision real number corresponding to the value of the first variable. No explicit type conversion is required and the offset array is defined in the GUI startup time rather than during run time so the conversion process is instantaneous.

Animation

Every effort was made to speed up the looping of the GUI program so that it had the potential to accept data snapshots from the CIM Unit at a rate appropriate for real-time operation. The screens and the functions of the graphical displays they were to contain were determined and implemented initially without regard to execution speed. Once running, ways to increase the execution speed were investigated. If the graphics depend on the value of variables from the simulation and their full range can be adequately represented by a limited number pictorial representations, graphics can be created in advance. For instance, on the mouse-sensitive graphics window of the valve screen of the ICS GUI (figure 8), the valve positions are animated so the stems appear to rotate to the correct angle. Originally this was accomplished by redrawing the circles, lines, and arcs—approximately ten different shapes—over a valve stem every time its angle changed, but this proved to be extremely slow, especially with six valve stems to update. In general, it is faster to draw a *raster* image than create a shape. A raster image is a

pattern of pixels captured and stored in an array. The flows through the valves are represented by either red or green, thus two sets of rasters were needed. The raster creation is part of the preprocessing or start-up of the GUI so the time it takes is not critical. Into a blank screen were inserted as many valves as could fit, each with the stem turned a slightly different amount from fully open to fully closed. This screen was rasterized and stored and the process was repeated with the other flow color. Once these rasters are created, the appropriate rectangle containing the valve stem from the rasterized image, corresponding to a particular valve angle, can be drawn over the appropriate valve stem in the valve screen using a single :draw-raster message. This drawing is done so quickly that the valves appear to turn with a smooth, continuous motion. Previously, using the shape-drawing technique, only one valve stem was updated each time through the read-and-update-data loop. By replacing the shape-drawing portion of the valve animation routine with a raster-drawing procedure, it was demonstrated that the update time could be reduced. Using rasters, all six valve stem angles can be updated in each loop and it is still faster than the other method. This type of animation is used when the GUI must have complete control over what is illustrated, such as with the valve angles which depend on measurements.

When repetitive movements or ones that are independent of variables' values are to be depicted, it is convenient to use *sprites*. Sprites are graphical objects which move across the screen based on some preset parameters such as speed and direction. The GUI and the programmer do not have to worry about a sprite's movement once it is set. The calculations required to move the sprite are done automatically by the CPU and no intervention by the user or GUI is required. However, since the CPU must perform calculations to move the sprite, there is a noticeable slowdown in the GUI's speed. This occurs only when a screen which contains sprites is displayed, otherwise the calculations are suspended. Thus, if a screen would benefit from being animated and the movements should repeat while the screen is displayed, and not a lot of other activity is taking place on the screen (such as if the number of variables to plot is small) sprites are a reasonable option.

CONCLUSIONS

A prototype object-oriented Graphical User Interface was developed to monitor the real-time operation of a control system in a laboratory test bed. The GUI can be used as a research tool to aid in the development of complex control systems. Through plots, animation, interactive graphics, and text, the user can watch the closed loop system's performance to any level of detail the developer chooses. The tools and techniques used to create the GUI are described to allow a similar system to be constructed or for additions to be made to the current one. Using the basic structure of the existing GUI as a template, a programmer can fairly easily write the code for a new GUI, albeit without detailed pictures. Programming the code for the pictures in the mouse-sensitive graphics window is by far the most time consuming aspect of the GUI development. The hints given for speeding up the GUI's execution are simple and, if used from the initial design phase, will not increase development time.

There are numerous aspects to consider when planning to build a GUI, many of which are discussed here. If the GUI is modular and can be tested in a stand-alone mode, the development is greatly simplified. The Texas Instruments Explorer II+ used in the ICS application provided

good graphics capability with convenient development tools, the networking capability and speed required for the ICS application, and a good interactive development environment.

APPENDIX: Lisp code used to create SSME frame

```
(defflavor mouse-sensitive-graphics-window () ; flavor of upper window in frame
           (w:basic-mouse-sensitive-items
            gwin:graphics-window))

(defvar my-constraint-list ; constraint list created by Constraint Frame Editor
        '((t (pane2 dummy3) ((pane2 0.48))
             ((dummy3 :horizontal (:even) (pane4 pane5) ((pane4 0.5))
                      ((pane5 :even)))))))

(defvar ssme-pane-list ; description of each window in the SSME frame
        '((pane5 gwin:graphics-window
                 :bottom 753
                 :left   512
                 :right  1023
                 :top    361
                 :label  "SSME VARIABLE PLOTS")
          (pane4 w:lisp-listener
                 :bottom 753
                 :left   1
                 :right  511
                 :top    361
                 :label "INTERACTIVE")
          (pane2 ics:mouse-sensitive-graphics-window
                 :item-type-alist ; item-type-alist is a list of mouse-sensitive
                 ; items, function to execute when item is selected, and text to
                 ; display at bottom left of screen when mouse is over item
                 ((hpotp-item (prepare-hoftp-window)
                   "HIGH-PRESSURE OXIDIZER TURBOPUMP")
                  (hpftp-tip-seal-item (prepare-hpftp-tip-seal-window)
                   "HIGH-PRESSURE FUEL TURBOPUMP")
                  (lpftp-item (prepare-lpftp-window)
                   "LOW-PRESSURE OXIDIZER TURBOPUMP")
                  (opov-item (prepare-valve-window)
                   "OXIDIZER-PREBURNER OXIDIZER VALVE")
                  (opfv-item (prepare-valve-window)
                   "OXIDIZER-PREBURNER FUEL VALVE")
                  (mov-item (prepare-valve-window) "MAIN OXIDIZER VALVE")
                  (ccv-item (prepare-valve-window) "CHAMBER COOLANT VALVE")
                  (mfv-item (prepare-valve-window) "MAIN FUEL VALVE")
                  (fpov-item (prepare-valve-window)
                   "FUEL-PREBURNER OXIDIZER VALVE")
                  (sensor-item (prepare-sensor-window) "SENSOR SCREEN"))
                 :bottom 361
                 :left   1
                 :right  1023
                 :top    1
                 :label "SPACE SHUTTLE MAIN ENGINE FULL VIEW"
                 :expose-p t)))

(defvar my-edge-list '(0 0 1024 754)) ; outside edges of the frame

(defun make-frame (pane-list constraint-list) ; function used to create a frame
  (make-instance 'w:bordered-constraint-frame
                 :save-bits t
                 :edges my-edge-list
                 :panes pane-list
                 :constraints constraint-list
                 :expose-p t))

(make-frame ssme-pane-list my-constraint-list) ; call to create SSME frame
```

REFERENCES

1. Litt, J. S.; et al, "An Object-Oriented Graphical User Interface for a Reusable Rocket Engine Intelligent Control System," NASA TM xxxx, 1994.

2. Simon, D. L.; Wong, E.; Musgrave, J. L., "Implementation of an Intelligent Control System," Proceedings of the Advanced Earth-to-Orbit Propulsion Technology Conference, Huntsville, AL, May 19-21, 1992.

3. DeLaat, J. C.; Soeder, J. F., "Design of a Microprocessor-Based Control, Interface and Monitoring (CIM) Unit for Turbine Engine Controls Research," NASA TM-83433, 1983.

4. Schlaer, S.; Mellor, S. J., *Object-Oriented Systems Analysis: Modeling the World in Data*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1988.

5. ——, Explorer™ Window System Reference, Texas Instruments Incorporated, Data Systems Group, Austin, TX, December 1987.

6. ——, Explorer™ LISP Reference, Texas Instruments Incorporated, Data Systems Group, Austin, TX, June 1987.

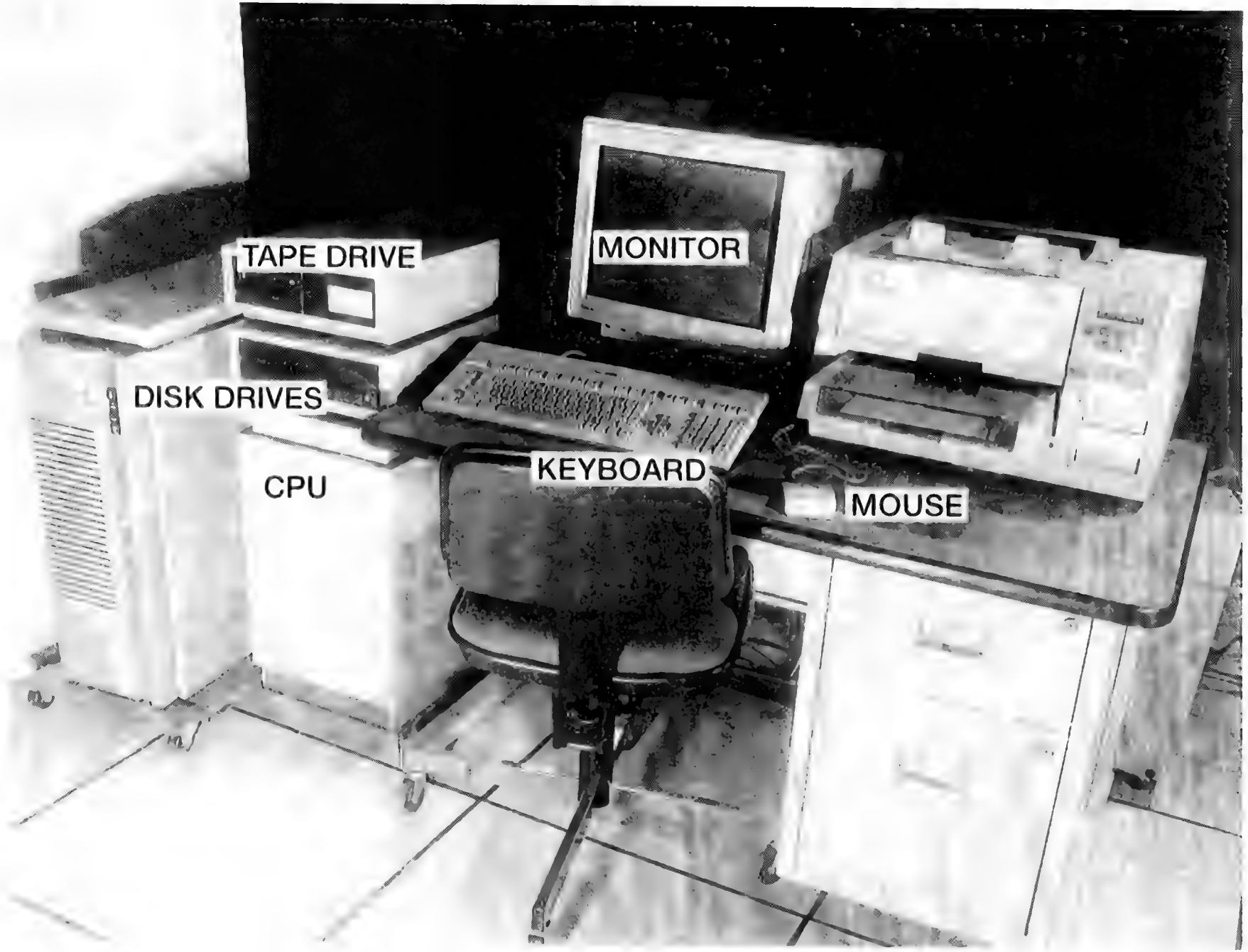

Figure 1. TEXAS INSTRUMENTS EXPLORER SYSTEM

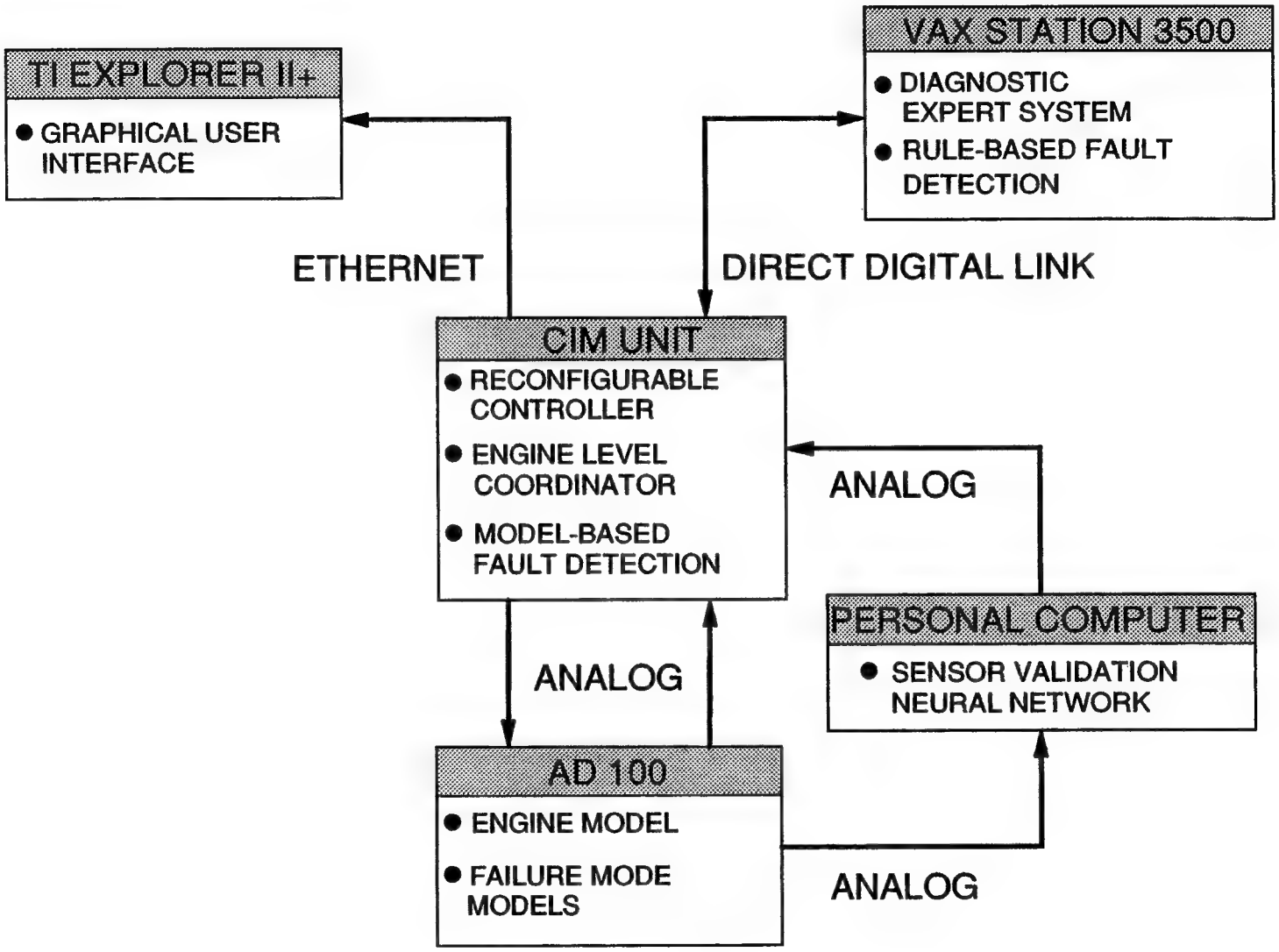

Figure 2. INTELLIGENT CONTROL SYSTEM SIMULATION TEST BED

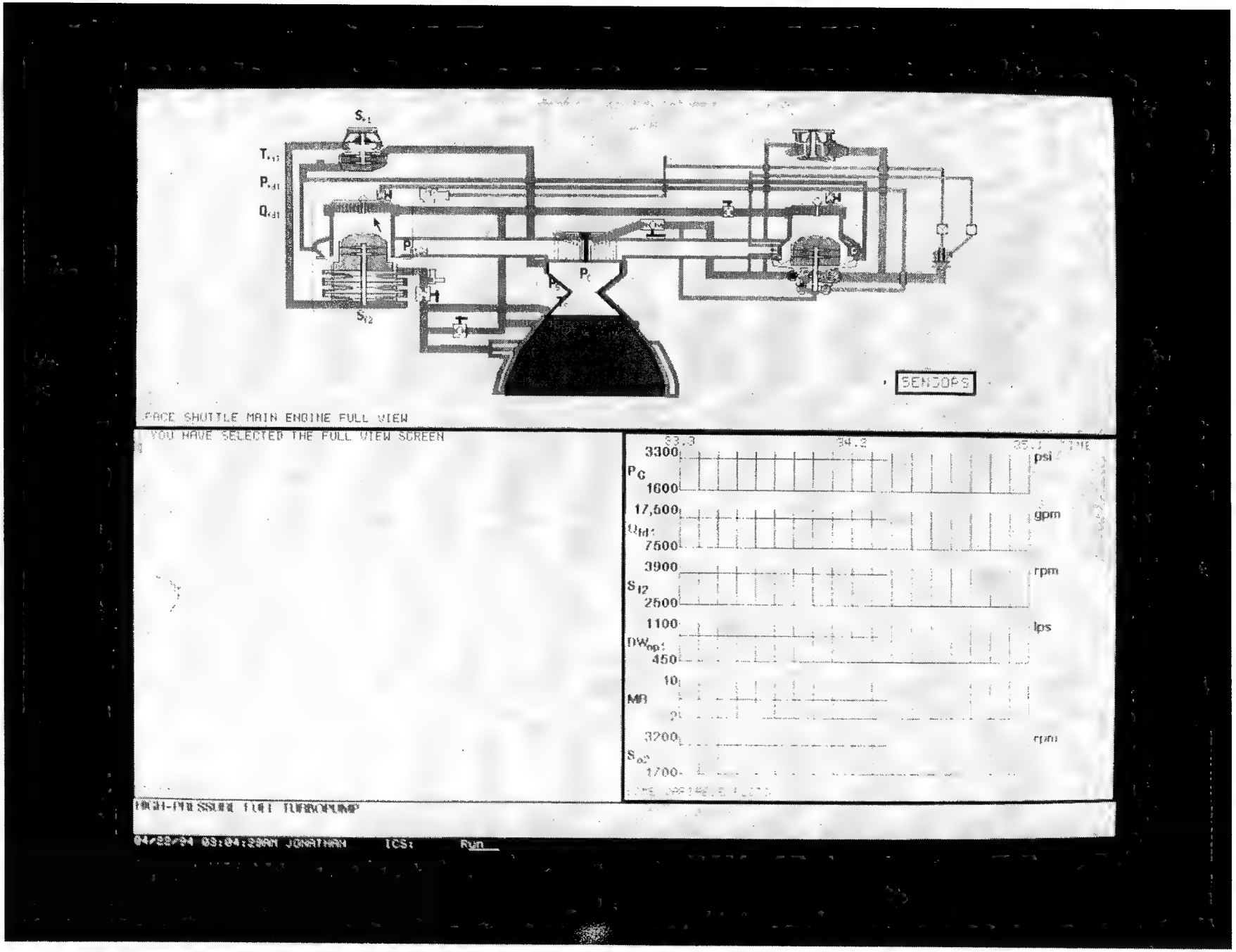

Figure 3. EXAMPLE OF A FRAME CONSISTING OF THREE WINDOWS

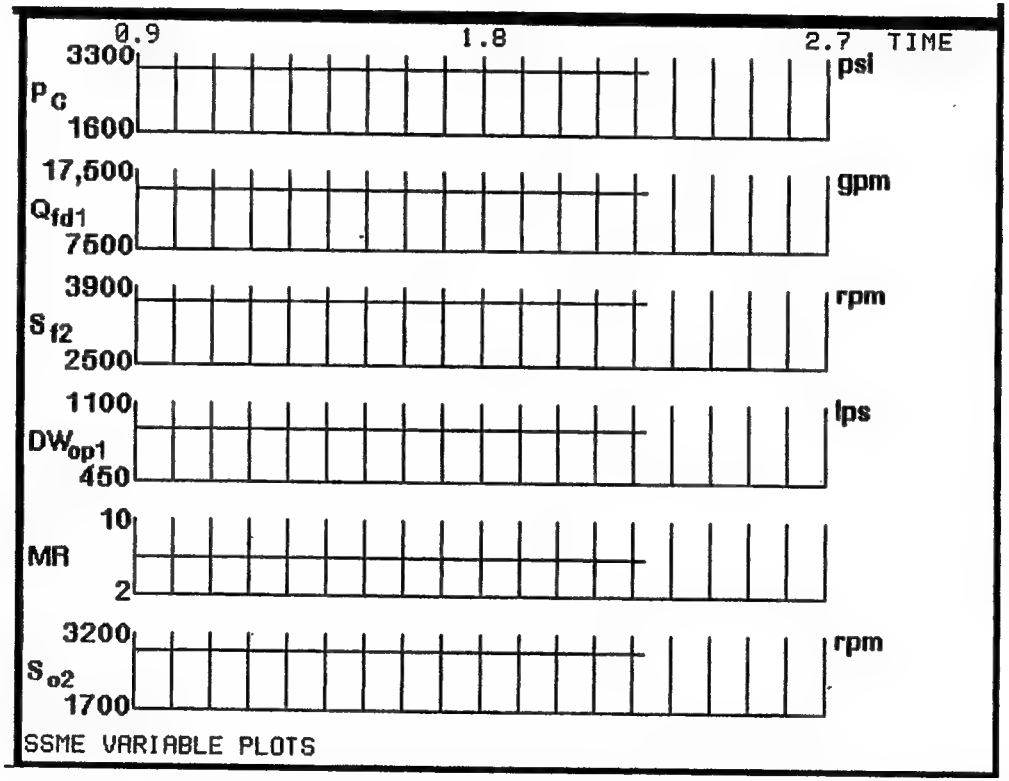

Figure 4. EXAMPLE OF A PLOTTING WINDOW WITH AXES AND DATA

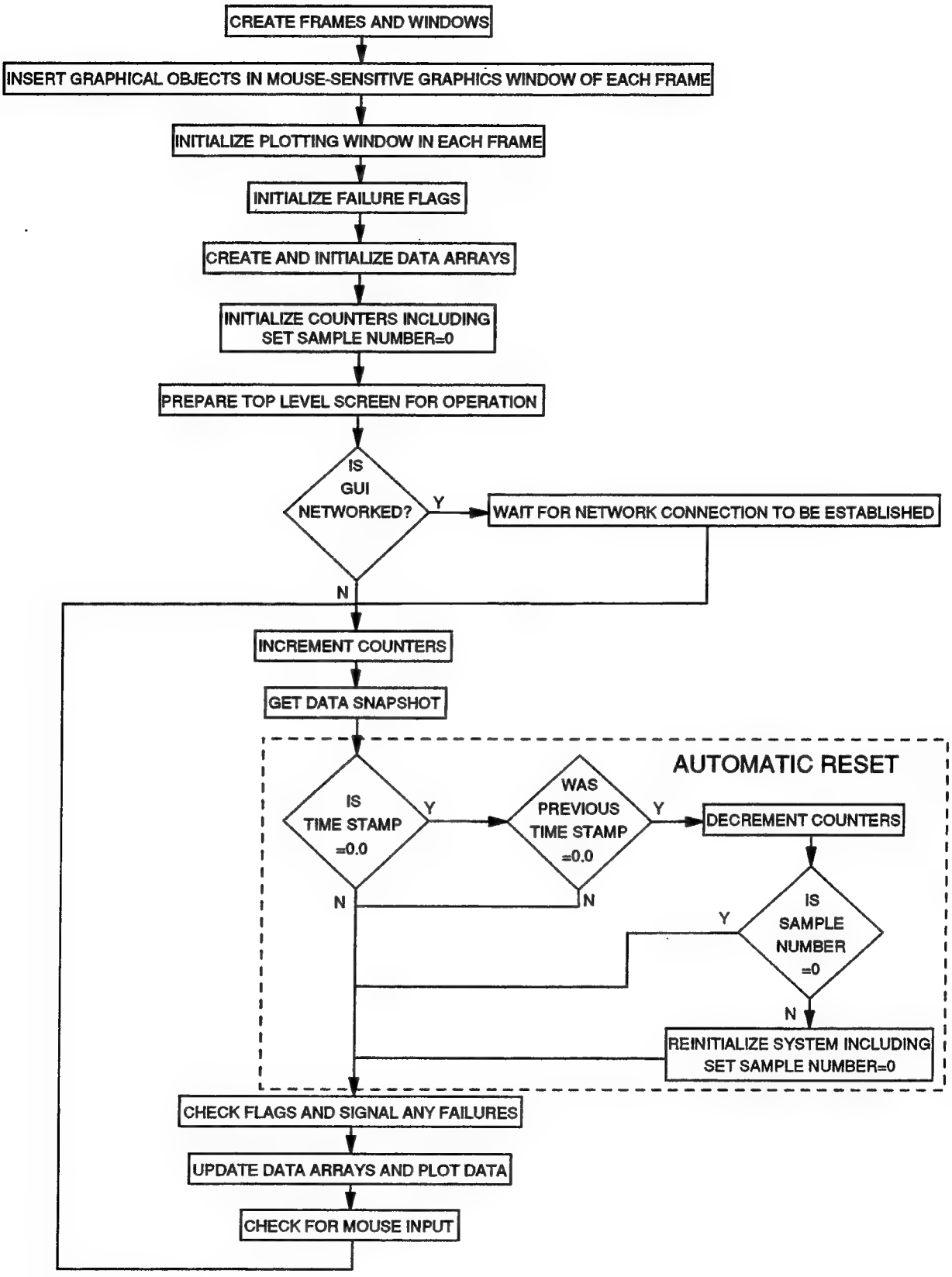

Figure 5. FLOW CHART FOR GUI

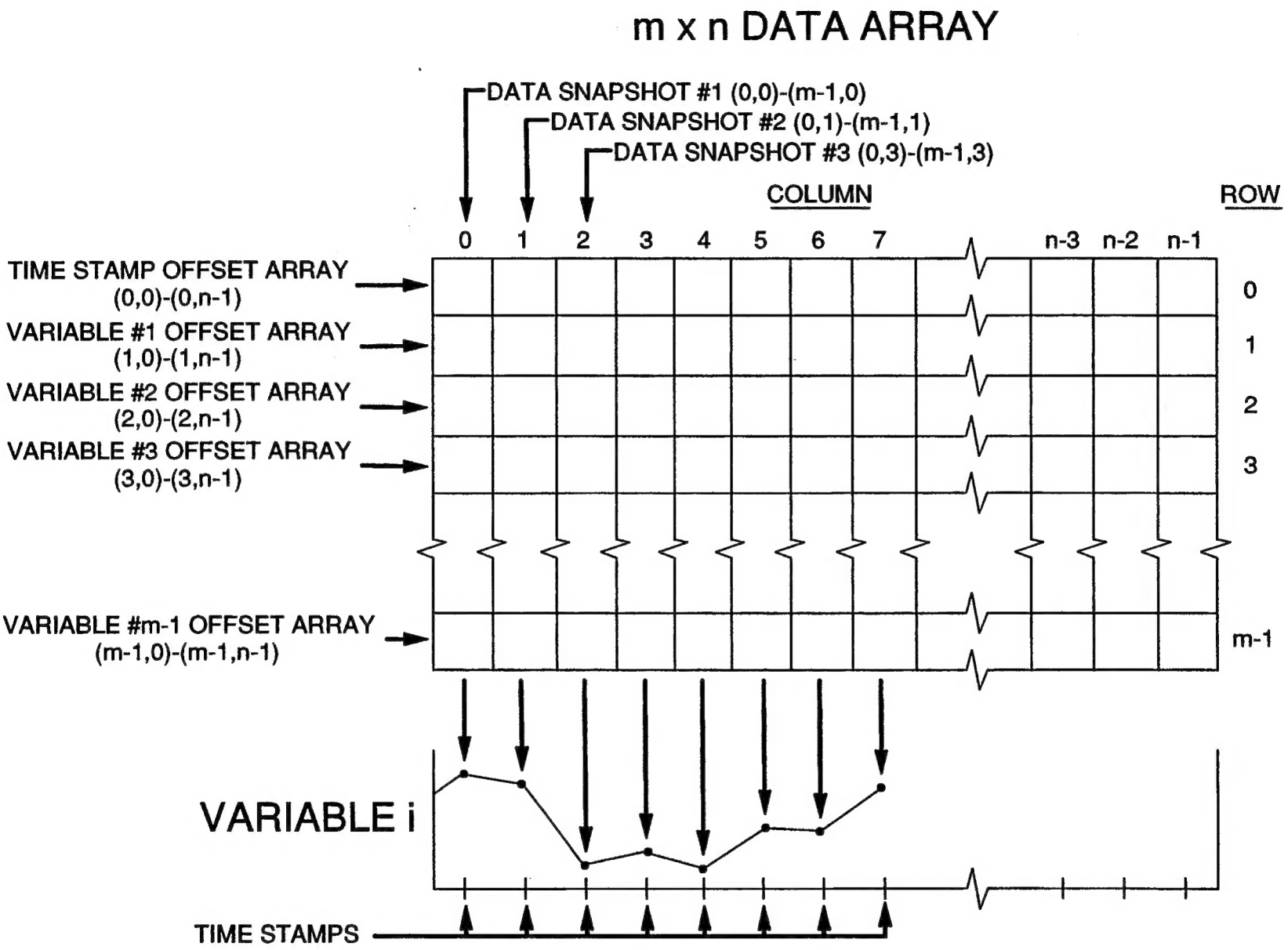

Figure 6. TWO-DIMENSIONAL ARRAY USED TO STORE DATA SNAPSHOTS

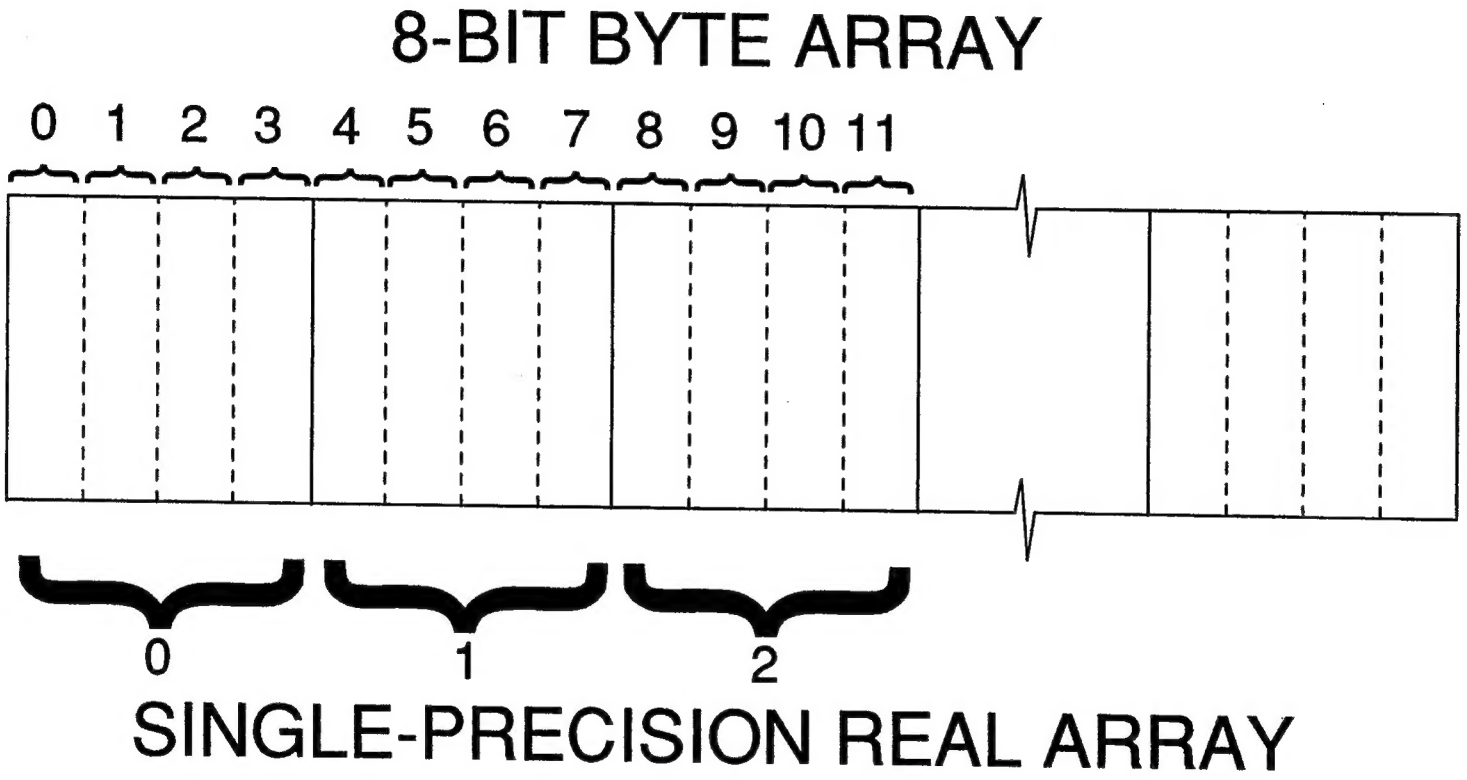

Figure 7. EXAMPLE OF TWO ARRAYS OF DIFFERENT TYPES SHARING COMMON MEMORY.

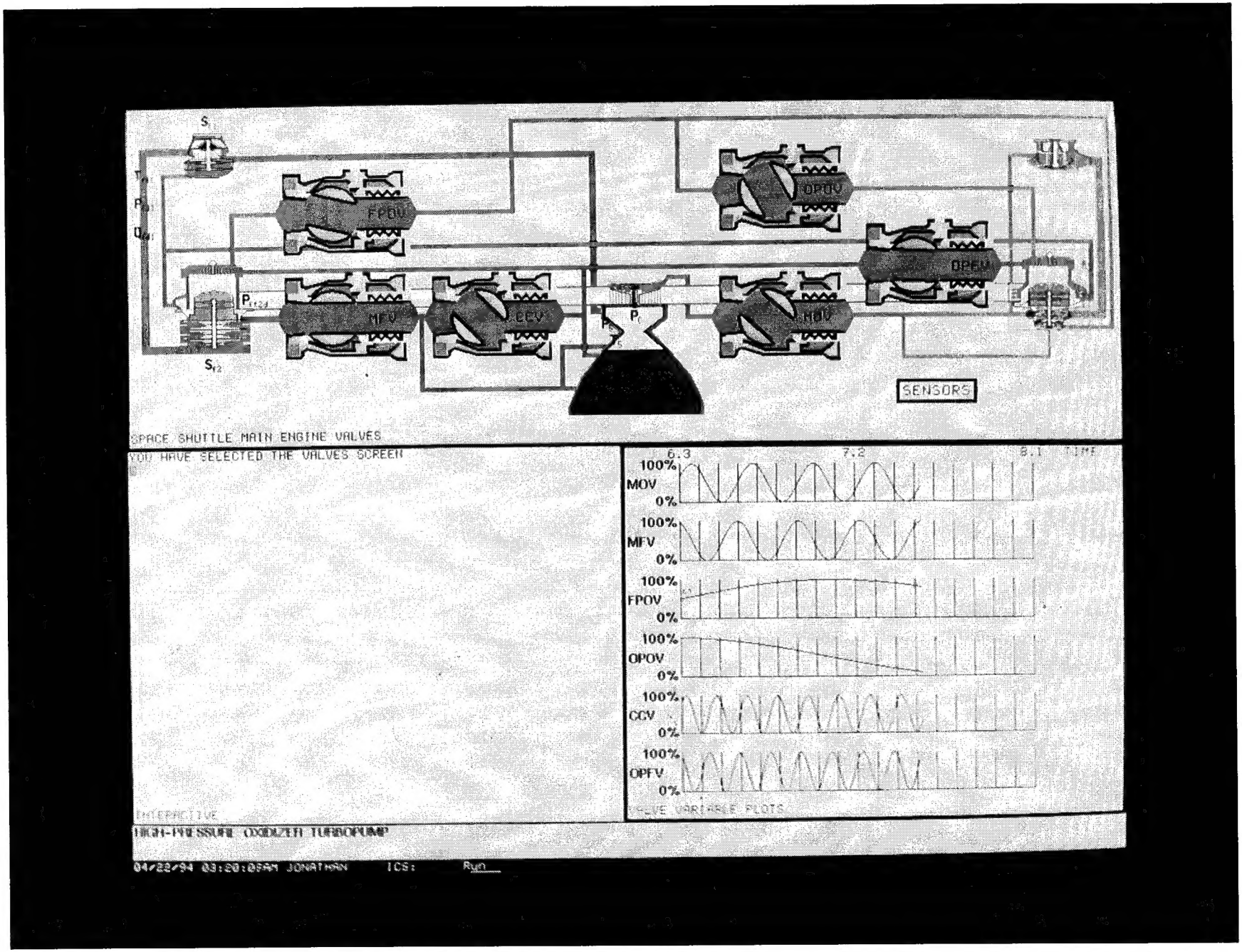

Figure 8. ICS VALVE FRAME WITH ROTATING VALVE STEMS

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 1994	Technical Memorandum

4. TITLE AND SUBTITLE

A Prototype Lisp-Based Soft Real-Time Object-Oriented Graphical User Interface for Control System Development

5. FUNDING NUMBERS

WU–468–01–11
1L161102AH45

6. AUTHOR(S)

Jonathan Litt, Edmond Wong, and Donald L. Simon

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

NASA Lewis Research Center
Cleveland, Ohio 44135–3191
and
Vehicle Propulsion Directorate
U.S. Army Research Laboratory
Cleveland, Ohio 44135–3191

8. PERFORMING ORGANIZATION REPORT NUMBER

E–9155

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, D.C. 20546–0001
and
U.S. Army Research Laboratory
Adelphi, Maryland 20783–1145

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

NASA TM–106743
ARL–TR–571

11. SUPPLEMENTARY NOTES

Jonathan Litt and Donald L. Simon, Vehicle Propulsion Directorate, U.S. Army Research Laboratory, NASA Lewis Research Center; Edmond Wong, NASA Lewis Research Center. Responsible person, Jonathan Litt, organization code 0300, (216) 433–3748.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified - Unlimited
Subject Category 60

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

A prototype Lisp-based soft real-time object-oriented Graphical User Interface for control system development is presented. The Graphical User Interface executes alongside a test system in laboratory conditions to permit observation of the closed loop operation through animation, graphics, and text. Since it must perform interactive graphics while updating the screen in real time, techniques are discussed which allow quick, efficient data processing and animation. Examples from an implementation are included to demonstrate some typical functionalities which allow the user to follow the control system's operation.

14. SUBJECT TERMS

Graphical user interface

15. NUMBER OF PAGES
23

16. PRICE CODE
A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	